# CSCE 221 Cover Page

# Programming Assignment #2

First Name Victoria Last Name Rivera Casanova UIN 927007294

User Name vrc E-mail address vrc@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more: Aggie Honor System Office

| Type of sources | | | |
|---|---|---|---|
| People | UGTA | | |
| Web pages (provide URL) | Piazza | StackOverflow | GeeksforGeeks |
| Printed material | C++ textbook | | |
| Other Sources | Class Notes and slides | | |

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work.
"On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work."

Your Name Victoria Rivera Casanova Date 6/17/2020

## Programming Assignment 2 (100 points)

Due June 16 at midnight to eCampus

1. Submission

(a) Zip the C++ files above and any additional files (input and output files).

i. (2 pts) Program description; purpose of the assignment.

1 This programming assignment is meant to get us comfortable implementing Minimum Priority Queues using 3 different data structures. We had to make 3 queues: a vector, linked list and binary heap. Each of these implementations have a different way of writing the common functions associated with a priority queue (remove_min(), is_empty(), insert()). There was a lot of freedom in how we were allowed to make our data structures. For example, we could sort the items on priority upon insertion or during removal. During this assignment, we also had to so a simulation of running time or a "time slice" for the 3 biggest text files. We also use a timing library/function to help verify our implementations based on the theoretical run times for our implementation functions. Another requirement is creating output files for our organized data and this requires File I/O.

2 **Running time BigO** of functions can be found in the header files for each individual function. In the main, I comment on how many times each function is getting called within their while loops.

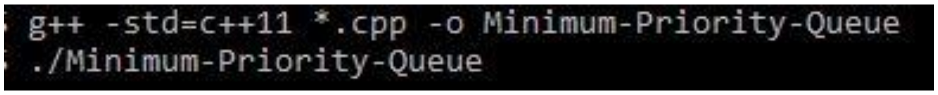
3 **Header Files:** "Binary.h", "Vector-imp.h", "LinkedList-imp.h"

4 **Main:** "mpq-main.cpp"

5 **Executable:** "Minimum-Priority-Queue"

ii. (2 pts) Instructions to compile and run your program; input and output specifications.

1 The instructions that were given were to run our code using c++ -std=c+=11 *.cpp -o pa2. Mine was a little different because I had my files in another folder titles 'Minimum-Priority-Queue', so my executable file is called Minimum Priority Queue and run by typing **./Minimum-Priority-Queue**. In our zip file we need at least 4 C++ files including the main. My test files are also located in this folder to be able to link up with my cpp files. The assignment document also gave us a specific way of outputting the list of our results. I declared it just how the document says in my code, plus, I added the statement "No more jobs to run" when a data structure was "empty" and all the values removed in order.

a 

```
g++ -std=c++11 *.cpp -o Minimum-Priority-Queue
./Minimum-Priority-Queue
```

2 **Data Files:** Output data files are provided inside the zip folder for all initial files provided. I used the Linked List implementation to read each file and write the formatted data in a new text file.

iii. (5 pts) Programming style, and program organization and design: naming, indentation, whitespace, comments, declaration, variables and constants, expressions and operators, error handling and reporting, files organization, operators overloading. templates. Please refer to the PPP-style document.

1 I organized all my implementations into their own header files and created helper functions to go with each one where I thought it necessary. For example, In the Linked List implementation I modified insert to call an insert_before() and insert_first() when the time was appropriate. This helped me for organization of my logic and code. The same is true for the "==" and ">" operator overloading. Then, in the main function I have 3 sections for the 3 implementation types. This includes reading from the files, inserting and removing in correct order. I read from the files within the main function without overloading the input operator. For all 3 header files I overloaded the output, less than and greater than operators. Inside the main function, I have 3 different sections for the 3 MPQ implementations.

2 Later on, UGTA Isaiah clarified how the binary Heap needed a Binary heap object (with a vector), so I had to change my implementation for the Binary Heap MPQ. So, I had to create a "wrapper" class for my binary heap object. That is where my confusion was. The wrapper class is found in BinanryHeap.h alongside the BinaryHeap object class.

3 Within the main function I separated the code into sections for Phase 1, Phase 2, and Phase 3. Where in Phase 1, I implement the vector mpq and linked list mpq using int as type T. In Phase 2, I implement the

Binary heap object and  mpq using int as type T. Finally, in Phase 3, I used all three implementations to take in type CPU_Job (a struct). Included in the LinkedList data collection, is the writing of the output files. I made a note of this as a comment in the code.

4 In my programs I use proper indentations for loops and other places where it's necessary. The way I timed it was by commenting out the other 2 and focusing on one implementation at a time. Then at the end, I ran the entire program all together and got similar numbers as the previous way. I followed the naming conventions of the functions given, but then I used CamelCase naming for my variable and other function declarations. As for whitespace, I try not to leave a lot, but I have at least one line separating major parts of functions from each other (loops, variable declarations etc). I use comments throughout my code to state the runtime for each function and I comment out many print statements that I used for testing purposes. I do go back and add comments to help in terms of readability. As far as error handling, I used the struct Empty List exception from pa1 and modified it for this assignment and threw it whenever a data structure was empty and there was a call to remove the min. ultimately I didn't need to use it because my while loop only ran until the end of file or while the data structure was filled (in other words, there was never a time that the remove function was called on an empty structure).

iv. (5 pts) Discussion of the implementation and running time in the terms of big-O.

1 From what I understood of the instructions, I created 3 different implementations for a Minimum Priority Queue with the private data member as the data structure (vector or linked list, for binary heap I used a vector to create said heap). And the member functions, like remove_min() and insert(), were tailored for specifically that data member.  I also left in struct objects I created for testing but have no bearing on the assignment instructions.

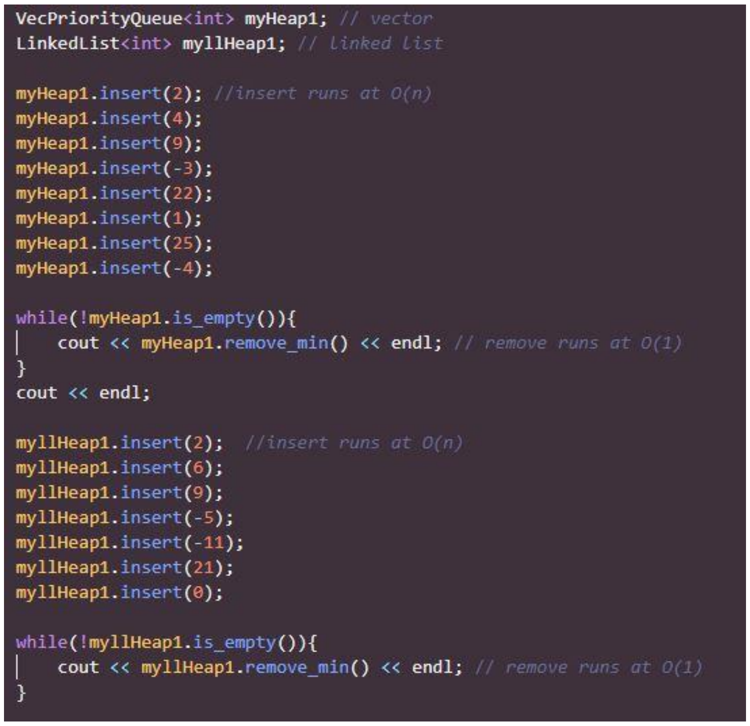
2 **Vector:** The insert function runs at O(1) time because I just push back the given object to the back. This allows for my remove function to run at O(n) time because it has to search the entire list to find the minimum value and the return said value.

3 **Linked List:** The insert function runs at O(n) (worst case) time because the nodes are sorted upon insertion and therefore, the remove function runs at constant O(1) time (it just has to remove the first node from the list). I modeled my Linked List as a doubly linked list with a header and trailer. I did try to insert the nodes first at the end and traverse the list to find the correct node when the remove function was called, but the node was not being deleted properly. (I do believe how I have it implemented now is logically simpler and it works at the appropriate theoretical time).

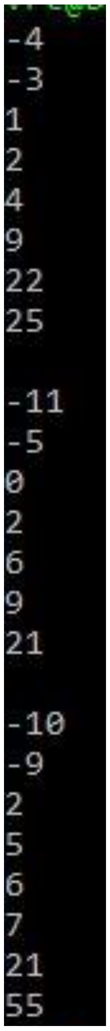
4 **Binary Heap:** The binary heap implementation has the fastest run time of all the implementations. It is based on a vector also. The remove function runs at O($log_2 n$) time. This is because, even though it only has to remove the root node, after the removal, the function performs walk-downs to reorganize the binary heap. The same principal applies to the insert function: it has a runtime of O($log_2 n$). The node is inserted at the end and must perform walk-ups to find its correct position within the heap. The maximum number of walk downs or walkups performed are $log_2 n$ as that is equivalent to the height of the binary heap. I created a Binary heap MPQ wrapper class to implement in Phase 3. Within this wrapper class "BinaryHeap", I have a binary heap object "BinaryHeapObj", from phase 2.

5 The insert() and remove() function are being called n times each and (n = 4, 10, 100,… etc) You have to insert each node and that takes n function calls and then you have to remove the data and that takes another n functions calls (depending on which file we are reading at the time).

## 6 Evidence of Testing for Phase 1-2:



```
VecPriorityQueue<int> myHeap1; // vector
LinkedList<int> myllHeap1; // linked list

myHeap1.insert(2); //insert runs at O(n)
myHeap1.insert(4);
myHeap1.insert(9);
myHeap1.insert(-3);
myHeap1.insert(22);
myHeap1.insert(1);
myHeap1.insert(25);
myHeap1.insert(-4);

while(!myHeap1.is_empty()){
|   cout << myHeap1.remove_min() << endl; // remove runs at O(1)
}
cout << endl;

myllHeap1.insert(2);  //insert runs at O(n)
myllHeap1.insert(6);
myllHeap1.insert(9);
myllHeap1.insert(-5);
myllHeap1.insert(-11);
myllHeap1.insert(21);
myllHeap1.insert(0);

while(!myllHeap1.is_empty()){
|   cout << myllHeap1.remove_min() << endl; // remove runs at O(1)
}
```



←Vector

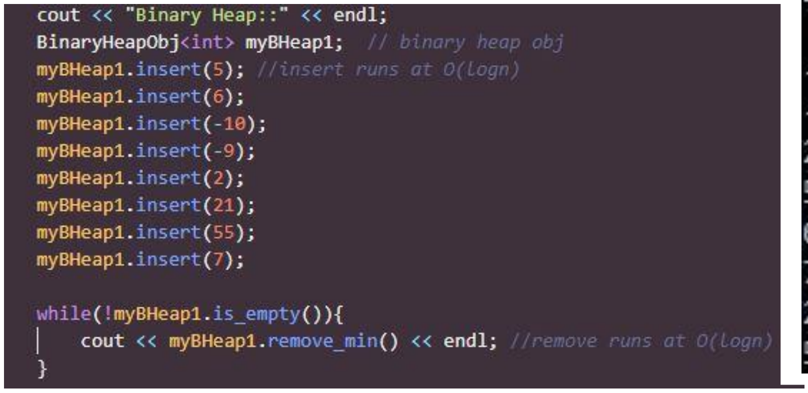
```
-4
-3
1
2
4
9
22
25
```

←Linked List

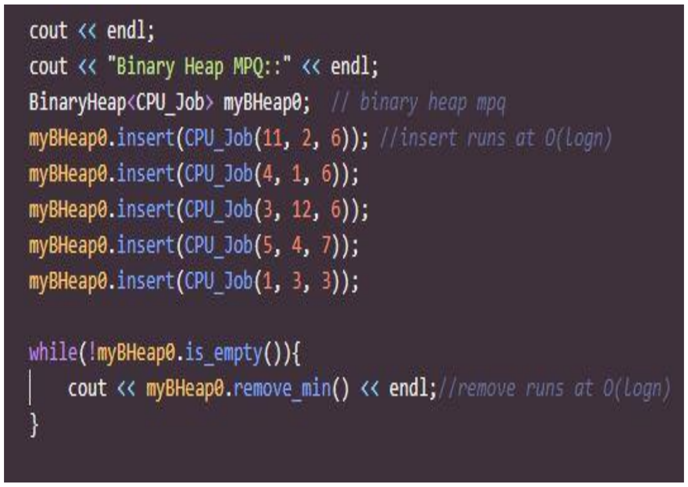
```
-11
-5
0
2
6
9
21
```

←Binary Heap Obj

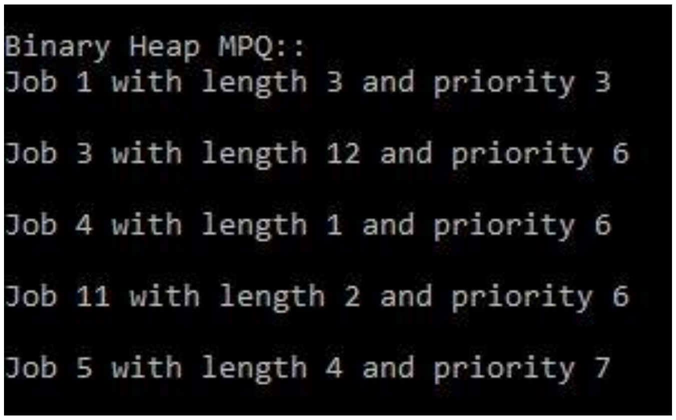
```
-10
-9
2
5
6
7
21
55
```



```
  cout << "Binary Heap::" << endl;
  BinaryHeapObj<int> myBHeap1;  // binary heap obj
  myBHeap1.insert(5); //insert runs at O(logn)
  myBHeap1.insert(6);
  myBHeap1.insert(-10);
  myBHeap1.insert(-9);
  myBHeap1.insert(2);
  myBHeap1.insert(21);
  myBHeap1.insert(55);
  myBHeap1.insert(7);

  while(!myBHeap1.is_empty()){
  |   cout << myBHeap1.remove_min() << endl; //remove runs at O(logn)
  }
```

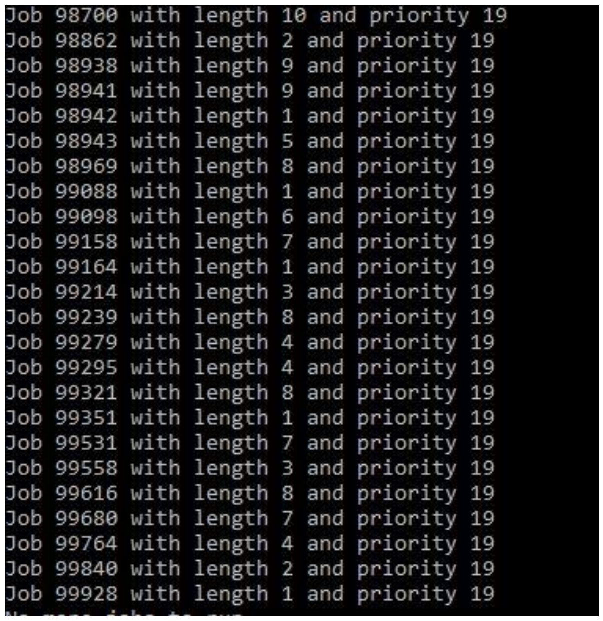


```
cout << endl;
cout << "Binary Heap MPQ::" << endl;
BinaryHeap<CPU_Job> myBHeap0;  // binary heap mpq
myBHeap0.insert(CPU_Job(11, 2, 6)); //insert runs at O(logn)
myBHeap0.insert(CPU_Job(4, 1, 6));
myBHeap0.insert(CPU_Job(3, 12, 6));
myBHeap0.insert(CPU_Job(5, 4, 7));
myBHeap0.insert(CPU_Job(1, 3, 3));

while(!myBHeap0.is_empty()){
|   cout << myBHeap0.remove_min() << endl;//remove runs at O(logn)
}
```



```
Binary Heap MPQ::
Job 1 with length 3 and priority 3

Job 3 with length 12 and priority 6

Job 4 with length 1 and priority 6

Job 11 with length 2 and priority 6

Job 5 with length 4 and priority 7
```

**Sample output from Phase 3:**

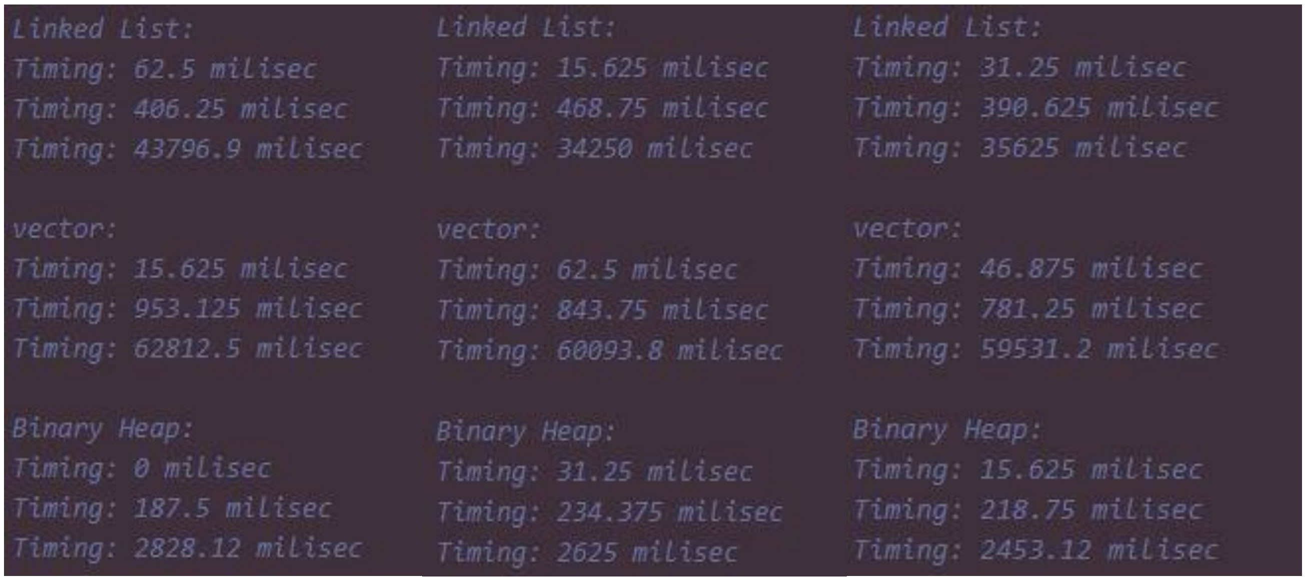


```
Job 98700 with length 10 and priority 19
Job 98862 with length 2 and priority 19
Job 98938 with length 9 and priority 19
Job 98941 with length 9 and priority 19
Job 98942 with length 1 and priority 19
Job 98943 with length 5 and priority 19
Job 98969 with length 8 and priority 19
Job 99088 with length 1 and priority 19
Job 99098 with length 6 and priority 19
Job 99158 with length 7 and priority 19
Job 99164 with length 1 and priority 19
Job 99214 with length 3 and priority 19
Job 99239 with length 8 and priority 19
Job 99279 with length 4 and priority 19
Job 99295 with length 4 and priority 19
Job 99321 with length 8 and priority 19
Job 99351 with length 1 and priority 19
Job 99531 with length 7 and priority 19
Job 99558 with length 3 and priority 19
Job 99616 with length 8 and priority 19
Job 99680 with length 7 and priority 19
Job 99764 with length 4 and priority 19
Job 99840 with length 2 and priority 19
Job 99928 with length 1 and priority 19
```

This is output code from Binary Heap MPQ of "SetSize100000.txt". The order is correct because since they have the same priority, then the algorithm compares the Job IDs. The lower the job ID, the higher the priority.

v. (6 pts) Presenting the testing results, use a table to present timings for each implementation (9 timings in total).

1 Raw times for 3 trials of each large file:



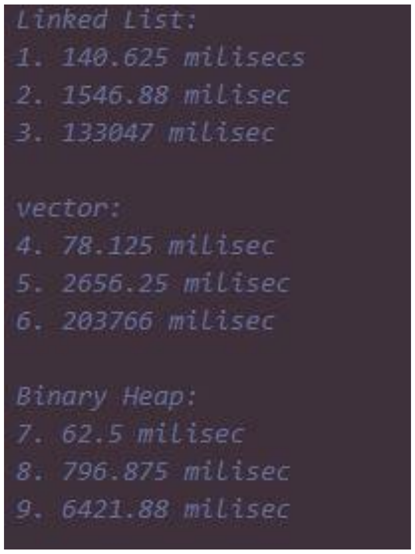
```
          Linked List:                   Linked List:                   Linked List:
1000->    Timing: 62.5 milisec           Timing: 15.625 milisec         Timing: 31.25 milisec
10000->   Timing: 406.25 milisec         Timing: 468.75 milisec         Timing: 390.625 milisec
100000->  Timing: 43796.9 milisec        Timing: 34250 milisec          Timing: 35625 milisec

          vector:                        vector:                        vector:
1000->    Timing: 15.625 milisec         Timing: 62.5 milisec           Timing: 46.875 milisec
10000->   Timing: 953.125 milisec        Timing: 843.75 milisec         Timing: 781.25 milisec
100000->  Timing: 62812.5 milisec        Timing: 60093.8 milisec        Timing: 59531.2 milisec

          Binary Heap:                   Binary Heap:                   Binary Heap:
1000->    Timing: 0 milisec              Timing: 31.25 milisec          Timing: 15.625 milisec
10000->   Timing: 187.5 milisec          Timing: 234.375 milisec        Timing: 218.75 milisec
100000->  Timing: 2828.12 milisec        Timing: 2625 milisec           Timing: 2453.12 milisec
```

What I timed specifically, using ctime, was 1) the insertion of each "node" into their unique data structures and 2) each remove function call. The 9 times that are called for are below:

| | 1000 size file | 10000 size file | 100000 size file |
|---|---|---|---|
| Linked List | **36.458** miliseconds | **421.875** miliseconds | **37890.533** miliseconds |
| Vector | **41.667** miliseconds | **859.375** miliseconds | **60812.5** miliseconds |
| Binary Heap | **15.625** miliseconds | **213.54** miliseconds | **2635.413** miliseconds |

**Analysis:** From the readings, we can see that the Binary Heap implementation takes less time than either the vector and Linked List implementations. This is accurate because its MPQ functions have a faster run time at O($log_2 n$) compared to the Vector and Linked List implementations. The increase in time as the files get bigger is also to be expected because you are making more function calls as you insert and remove a greater number of values. From the data, it seems the Linked List MPQ runs faster than the Vector MPQ. The discrepancy between the Linked List and Vector times can be due to the fact that since the vector implementation is unsorted upon insertion, it is always going to run O(n) in the removal function (it has to check each value for the minimum). While the linked list since it is sorted upon insertion, in some cases it won't have to traverse the whole list (it stops once there is a node that is greater than itself and is inserted before this node). **Theoretically, they still have a worst case of O(n) (in insert for Linked List and remove for Vector), based on my implementation.**

There can also be some errors in timing based on other programs the computer is running. The first time I tried to run the timings, I was watching a video as the same time and those output times were slower than the official ones included in this report:



```
Linked List:
1. 140.625 milisecs
2. 1546.88 milisec
3. 133047 milisec

vector:
4. 78.125 milisec
5. 2656.25 milisec
6. 203766 milisec

Binary Heap:
7. 62.5 milisec
8. 796.875 milisec
9. 6421.88 milisec
```

You can clearly see these times are slower than the previous ones

It really depends on what your computer is doing in the background to get a somewhat accurate reading on the function times.